Homework 3: Q2

Name: Waiwai Kim, waiwaiki

1 Part (a): Algorithm Details

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_1 + x_2 + x_3 \\ 0 + x_2 + x_3 \\ 0 + 0 + x_3 \end{bmatrix} \tag{1}$$

For part (a) of Question 2, we will leverage the algorithm that we have already built for Question 1. First, let's review the algorithm used to efficient computer multiplication between a unit upper triangular matrix and an input vector. The algorithm idea is based on the output's pattern that the last element is the last element of the input vector, that the second element is the sum of the second element of the input vector and and its previous output element and so on. Equation 1 shows an example when $n = 3$. Note that, if you reverse-traverse from $i = 2$ to $i = 0$, the output element gets "accumulated". In other words, element$_i$ = element$_{i-1}$ + input vector$_i$. The algorithm used for Question is presented in Algorithm 1.1. Note that this an actual Python code submission.

In line 4 of Algorithm 1.1, i iterates from $n-1$ to -1 (exclusive of -1) as i decrements by 1. Note that this enables the algorithm to reverse iterate. Line 6 checks whether it's the first element in the return vector. If it is, the algorithm simply adds the input element of x_n to the output vector. If it's not it will add the current input element of x_i to the previous element of the output vector. Once the algorithm completes the reverse traverse, the algorithm terminates.

Algorithm 1.1: Algorithm for Q1

```
1
2  return_vector = [0]*len(self.in_vector)
3
4  for i in range(len(self.in_vector)-1,-1,-1):
5
6          if i == (len(self.in_vector)-1):
7                  return_vector[i] = self.in_vector[i]
8
9          else:
10                 return_vector[i] = return_vector[i+1] + self.in_vector[i]
11
12 return return_vector
```

Now let's expand the aforementioned idea to apply to Question part(a) in which the upper triangular matrix is different integer . Looking at Equation 2, we notice that that the structured upper triangular matrix has a uniform number each row. For example, in row 2, all non-zero elements are 17. The effect of having such structure is similarly reflected in the multiplication output. Note that in the multiplication output in Equation 2, each element has one coefficient. For example, in row 2, both x_2 and x_3 have the same coefficient of 17. In row 1, all elements have the coefficient of 6.

$$\begin{bmatrix} 6 & 6 & 6 \\ 0 & 17 & 17 \\ 0 & 0 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6x_1 + 6x_2 + 6x_3 \\ 0 + 17x_2 + 17x_3 \\ 0 + 0 + 9x_3 \end{bmatrix} \tag{2}$$

The algorithm idea developed in Algorithm 1.1 can still be used. The algorithm needs to be adjusted so that it "swaps" the coefficients. As an example, we will focus on advancing from row 2 of the output vector to the row 2. Note that $17x_2 + 17x_3$ can be expressed as $17(x_2 + x_3)$, while, in row 1, $6x_1 + 6x_2 + 6x_3$ can be expressed as $6(x_1 + x_2 + x_3)$. Replying on the distributive law, we follow the steps below:

- dividing $17(x_2 + x_3)$ by 17 results in $(x_2 + x_3)$
- adding x_1 to $(x_2 + x_3)$ results in $(x_1 + x_2 + x_3)$
- multiplying $(x_1 + x_2 + x_3)$ results in $6x_1 + 6x_2 + 6x_3$

The algorithm presented in Algorithm 1.2 exactly does it. Note that the code is still written in Python and is a slight change to Algorithm 1.1.

Algorithm 1.2: Algorithm for Q2 Part (a)

```
1  '''
2  input 1 : in_vector = x
3  input 2 : r i.e) [6,17,9]
4  '''
5
6  for i in range(len(self.in_vector)-1,-1,-1):
7
8          if i == (len(self.in_vector)-1):
9                  return_vector[i] = self.in_vector[i] * r[i]
10
11         else:
12                 return_vector[i] = (return_vector[i+1]/r[i+1] + self.in_vector[i]) * r[i]
13
14 return return_vector
```

In line 6, for loop reverse traverses from $[n-1, 0]$. Line 8 checks if it's the first element. Line 6 to 8 have not changed. Line 9 adds the first element to the output after multiplying it by last element of the input vector of r. For example, $invector[n-1]$ is multiplied by $r[n-1]$, and the product is added to the bottom of the output. In line 12, we would still add the previous element of the output after removing the previous coefficient, which is why we divide the previous element by $r[i+1]$. Note that the algorithm reverse traverses, so $i+1$ precedes i. Once we get rids of the previous coefficient, we can add the current element. Finally, the sum can be multiplied by the current coefficient and be added to the output. The algorithm terminates when the for loop iterates to 0.

2 Part (a): Big-Oh Runtime Analysis

Lemma 2.1. *$T(n)$ for Algorithm 1.2 is $O(n)$*

We will show Lemma 2.1 is true based on the idea that the runtime of Algorithm 1.2 is dominated by the for loop in line 6, which always runs n times, and each iteration is $O(1)$.

In details, first note that there are always n iterations in the for loop in line 6. There is no early termination inside the for loop; thus, the for loop always executes the entirety of $[n-1, 0]$. Further, per each iteration of i, line 8, 9 and 12 can be implemented in $O(1)$. Line 8 involves a simple comparison. In Line 9, accessing a list with an index of i and multiplication can be done in $O(1)$. Also, assigning a value to an empty list is done in $O(1)$. Thus, we conclude that

Line 9 is done in $O(1)$. Similarly, we can conclude that the same amount of computation time is spent in Line 12.

$$T_{8-12} \leq O(1) = O(1) \tag{3}$$

Further, by the product lemma presented in the supporting material, the total time taken overall in line 6 to 12 is given by

$$T_{6-12} \leq O(n \cdot 1) = O(n) \tag{4}$$

Lastly, the line 14 is a simple return, so line 14 takes $O(1)$. Thus, the total time for the algorithm is given by:

$$T_{Algorithm1.2} = T_{6-12} + T_{14} \leq O(n) + O(1) \leq O(n) \tag{5}$$

Note that we rely on the sum lemma presented in the supporting material. Thus, we have shown that the Algorithm 1.2 has a runtime of $O(n)$.

3 Part (b): Algorithm Idea

The majority of thought process has already been explained in Part (a). In summary, we will leverage the algorithm 1.1 that was used for multiplication of a unit upper triangular matrix. Assume this algorithm works.

Now let's look at Equation 6 in order to formulate the pattern. Similar to part(a) where each row has a uniform number, we now have another type of structured upper triangular matrix of which each column has a uniform number. The effect of having such structure is similarly reflected in the multiplication output. Note that in the multiplication output in Equation 6, each element is an accumulated sum of the previous element and an current element with a current coefficient. For example, for the second iteration, the current element of the input vector, x_2 is multiplied by the current coefficient of 17. The product of $17x_2$ is added to the previous element of $9x_3$. In other words, it is very similar to algorithm 1.1, but each addition has to be adjusted with a respective coefficient.

$$\begin{bmatrix} 6 & 17 & 9 \\ 0 & 17 & 9 \\ 0 & 0 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6x_1 + 17x_2 + 9x_3 \\ 0 + 17x_2 + 9x_3 \\ 0 + 0 + 9x_3 \end{bmatrix} \tag{6}$$

4 Part (b): Algorithm Details

Again the majority of the thought process is very similar to Part(a). In line 6, for loop reverse traverses from $[n-1, 0]$. Line 8 checks if it's the first element. Line 9 adds the first element to the output after multiplying it by last element of the input vector of r. For example, $invector[n-1]$ is multiplied by $r[n-1]$, and the product is added to the bottom of the output. In line 12, we sum the previous element of the output and the current element multiplied by the current element. The sum is now added to the output. The algorithm terminates when the for loop iterates to 0.

Algorithm 4.1: Algorithm for Q2 Part (b)

```
1 '''
2 input 1 : in_vector = x
3 input 2 : r i.e) =[6,17,9]
4 '''
```

```
5
6  for i in range(len(self.in_vector)-1,-1,-1):
7
8          if i == (len(self.in_vector)-1):
9                  return_vector[i] = self.in_vector[i] * r[i]
10
11         else:
12                 return_vector[i] = return_vector[i+1] + (self.in_vector[i] * r[i])
13
14 return return_vector
```

5 Part (b): Big-Oh Runtime Analysis

Lemma 5.1. *$T(n)$ for Algorithm 1.2 is $O(n)$*

We will show Lemma 5.1 is true based on the idea that the runtime of Algorithm 4.1 is dominated by the for loop in line 6, which always runs n times, and each iteration is $O(1)$.

In details, first note that there are always n iterations in the for loop in line 6. There is no early termination inside the for loop; thus, the for loop always executes the entirety of $[n-1,0]$. Further, per each iteration of i, line 8, 9 and 12 can be implemented in $O(1)$. Line 8 involves a simple comparison. In Line 9, accessing a list with an index of i and multiplication can be done in $O(1)$. Also, assigning a value to an empty list is done in $O(1)$. Thus, we conclude that Line 9 is done in $O(1)$. Similarly, we can conclude that the same amount of computation time is spent in Line 12.

Further, by the product lemma presented in the supporting material, the total time taken overall in line 6 to 12 is given by

$$T_{6-12} \leq O(n \cdot 1) = O(n) \tag{7}$$

Lastly, the line 14 is a simple return, so line 14 takes $O(1)$. Thus, the total time for the algorithm is given by:

$$T_{Algorithm4.1} = T_{6-12} + T_{14} \leq O(n) + O(1) \leq O(n) \tag{8}$$

Note that we rely on the sum lemma presented in the supporting material. Thus, we have shown that the Algorithm 4.1 has a runtime of $O(n)$.

6 Part (b): Big-Omega Runtime Analysis

Note that the Big-Omega runtime analysis presented in the Supporting Material adopts a specific input family that forces the for loop to iterate n times. This methodology was used because the specific example algorithm can terminate prior to iterating n times.

In order to prove the lower bound of Algorithm 4.1, we have to notice the fact that this algorithm always runs through n iterations. There are always n iterations in the for loop in line 6. There is no early termination inside the for loop; thus, the for loop always executes the entirety of $[n-1,0]$. Thus we can say the following:

$$T_{6-12} \geq \Omega(n \cdot 1) = \Omega(n) \tag{9}$$

$$T_{Algorithm4.1} = T_{6-12} + T_{14} \geq \Omega(n) + \Omega(1) \geq \Omega(n) \tag{10}$$

Thus, without relying on a specific input family, we can say that the tight lower bound of the algorithm is $\Omega(n)$.

We have shown that the following:

$$T_{Algorithm4.1} \leq O(n)$$

$$T_{Algorithm4.1} \geq \Omega(n)$$

We can conclude that:

$$T_{Algorithm4.1} \geq \Theta(n)$$